Application of multimode HLS techniques to ASIP synthesis

Nina Engelhardt

Master Informatique spécialité Recherche en Informatique
Magistère Informatique et Télécommunications
ENS Cachan Antenne de Bretagne

Encadrant: Steven Derrien
Équipe CAIRN, IRISA

June 4, 2010

Abstract

Automatic generation of ASIPs is still insufficiently resource-efficient compared to human design. The current best effort relies on extending existing processor cores with custom instruction pathways, an approach that has several drawbacks: resources already existing in the core cannot be reused, and data access is restricted to the original register reads and writes, creating a bottleneck in throughput.

We present an ASIP synthesis algorithm that turns a semantic description of an instruction set into a synthesizable description of a processor's datapath in an efficient manner, based on a modification of Moreano's datapath merge algorithm. Support for operator mobility across pipeline stages, combinatorial loop prevention and realistic multiplexer resource usage estimation for FPGAs is added.

The translation chain is not yet completed, but preliminary results show efficient resource reuse between instruction datapaths.

Contents

Introduction **2**

1 Problem Statement **4**
1.1 ASIP Design Flow 4
1.2 High-Level and Multi-Mode Synthesis 7

2 Existing Approaches **10**
2.1 Adapting High-Level Synthesis Algorithms 10
2.2 Longest common substring 11
2.3 Pattern detection 11
2.4 Trimming 13
2.5 Datapath Merging 14

3 Proposed Modifications **17**
3.1 Timing Conditions on Mappings and Compatibility 17
3.2 Combinatorial Loop Detection 20
3.3 Multiplexer Weight Calculation 20
3.4 Operator Decomposition 22

4 Implementation **24**
4.1 Meta-Models 24
4.2 The Generation Workflow 25

5 Experimental Results **28**
5.1 Preliminary Results 28
5.2 Future Experiments 28

Conclusion **30**

Bibliography **31**

Glossary **33**

Appendix **35**

Introduction

Systems-on-Chip (SoC) are everywhere today, from the places where you would expect them (your DVD player) to the places you wouldn't (your washing machine).

A SoC is an integrated circuit containing several discrete elements that were previously on separate chips, such as a processing core, several specialized coprocessors, a communications unit, memory, etc. Integrating them on a single circuit has performance and power efficiency benefits.

SoC design poses several challenges. SoCs have to meet many constraints: they have to deliver high performance, often while being severely limited in power and area. They need to be developed quickly, as with many applications time-to-market is crucial. They also need to be bug-free, as it is often very costly or even impossible to modify an embedded system after its production.

To add the possibility of modification, or reconfigurability, to the system, the designer can use several building blocks which will ensure flexibility, at the cost of performance and elevated power consumption: Coarse Grain Reconfigurable Architectures, general-purpose CPUs, Digital Signal Processors, Application-Specific Instruction-set Processors (ASIP). Each of these building blocks represents a different degree of trade-off between flexibility and efficiency, as shown in figure 1. Target implementation technology may ultimately be ASIC or FPGA.

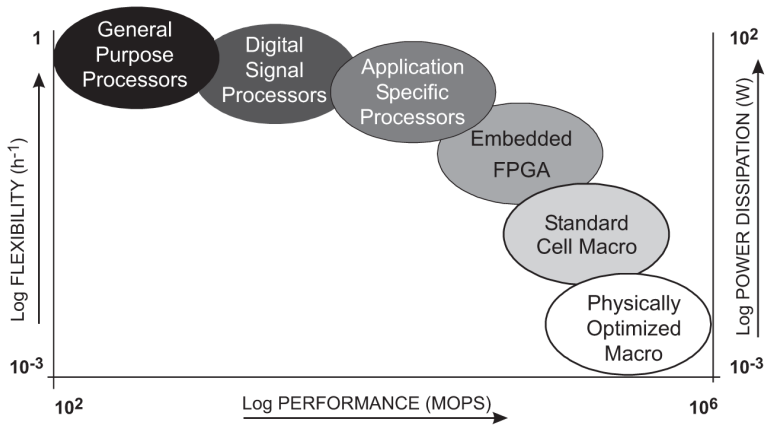


Figure 1: Comparison of different building blocks w.r.t. flexibility, performance, and energy efficiency [Image: T. Noll, EECS]

This work concerns Application-Specific Instruction Set Processors. As the name indicates, an ASIP is a processor whose instruction set is adapted to one specific algorithm or program. ASIPs present a good trade-off between flexibility and performance, and are especially well

suited for implementation in FPGA technology.

While the process of extracting an instruction set from an algorithm has been efficiently automated, the current state of automatic hardware generation for ASIPs still results in costly systems (in terms of surface) compared to human designs. Efficient solutions so far rely on extending existing processor cores, which introduces a number of constraints that limit opportunities for resource sharing.

We present a new approach to ASIP datapath generation based on *datapath merging*. Chapter 1 will present the datapath generation problem. Existing solutions, among which datapath merging, and their limitations are discussed in chapter 2. Our approach is described in chapter 3, and chapter 4 will discuss some implementation details. Finally some preliminary results will be shown in chapter 5.

Chapter 1

Problem Statement

This chapter presents the central problem motivating our work. A more detailed description of the concepts presented here and their related work follows in the subsequent chapters.

In section 1.1 we will present the different steps in ASIP development, from the initial algorithm to the custom hardware. Section 1.2 will touch upon the domains of high-level and multi-mode synthesis, which treat problems very similar to ours.

1.1 ASIP Design Flow

The aim of ASIP design is to produce both a hardware design of a processor and a software executable running on it from an algorithm description, generally in C. An overwiew of the design flow is shown in figure 1.1, and more detail is provided in the following subsections.

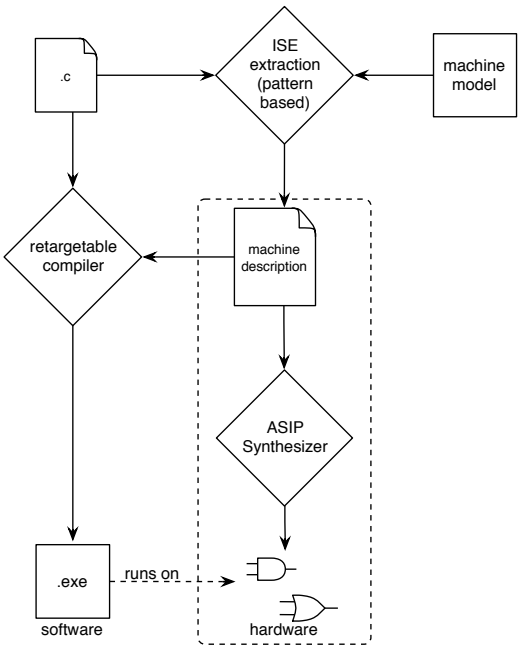
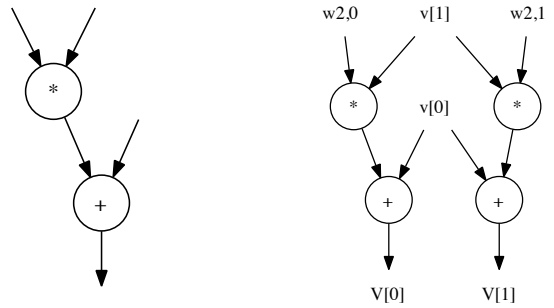


Figure 1.1: ASIP design flow. Marked area: where our work is situated.

Before customization, the architecture of an ASIP usually resembles a simple general-purpose processor of the RISC variety, sometimes with several parallel execution slots (VLIW). In addition to the general-purpose pipeline and simple instructions, interfaces are provided to extend both the controller and the execution pathways, so as to be able to add custom instructions to the processor. An example of such a processor is Altera's Nios II softcore[1]. In these custom instructions, one can implement complex patterns that occur frequently in the application. By exploiting the independence of certain operations to execute them in parallel, and simply by virtue of not traversing all pipeline stages for each individual elementary operation, performance can be greatly improved.

1.1.1 ISE Extraction

ASIPs are developed in two stages: first, in the ISE extraction step, the application source code is analyzed to detect computation patterns that appear frequently. If the application presents a great regularity, a few patterns can cover a large portion of the calculations: for instance, a FIR filter consists of a series of multiplications and additions, and implementing a multiply-add instruction can greatly speed up its execution. Similarly, a fast fourier transform has a basic pattern called "FFT butterfly" (cf. fig. 1.2).



(a) Multiply-add (b) FFT Butterfly

Figure 1.2: Examples of common patterns

The best suited patterns are used to define a custom instruction set. Several criteria are taken into account to determine which patterns to select: the frequency with which it appears during the execution of the application, how many operations it contains (the more the better), how long its critical path is (it has to fit within a pre-determined number of clock cycles), etc. This instruction set is then used with a processor model in order to create a *processor description* characterizing the instruction set and architecture, whose usage is twofold: To inform a retargetable compiler of the architecture it needs to compile for, and to synthesize an actual hardware implementation of this architecture.

1.1.2 Processor Description Languages

Unfortunately, the two uses the *processor description* has to serve are somewhat in conflict. Processor description languages exist for various purposes, depending on which they have different features [10]. The abstractions a compiler works on are generally *behavioural* models,

specifying the semantics of each instruction and some timing information, but no structural details. This is a problem especially with pipelined architectures, which are very hard to generate automatically.

Languages aimed at hardware synthesis such as *MAML*[10] therefore take a much closer look at the micro-architecture, describing separately each functional unit, each bus, each memory element. At this low level, describing a processor becomes a very time-intensive process requiring a great deal specialist knowledge. Some languages have been created to bridge the gap between the level of abstraction that is sufficient to generate a compiler and the level of detail needed to generate the architecture, such as *RADL* [12], but they do not deal with hardware reuse.

We will use a processor description language that is part of *PSTK*, an Xtext/Eclipse EMF-based tool developed by the CAIRN team. This language has semantic descriptions of the instruction set, but also some architectural elements: a pipeline is defined and memory and register reads and writes are placed within it. It is aimed at describing RISC processor architectures, and supports VLIW descriptions (our work, however, concerns only single-pipeline architectures). See appendix for an example of a PSTK processor description.

1.1.3 Hardware Synthesis

In a second step, a synthesizable processor architecture description implementing this instruction set is generated. While much work has been carried on by the compilation community in order to fully automatize the compilation step, current automatic solutions for this second step still have margin for improvement. Because automatically generating a processor from scratch is difficult, the current solution is to add custom datapaths to an existing design. For instance, Wolinski et al.[14] proposed a solution that generates datapaths for custom instructions that can be added to Altera's Nios II softcore. While a good customization interface such as the Nios' can offer a reasonable amount of flexibility, up to letting multi-cycle custom instructions stall the pipeline[1], it is still very constraining compared to a fully customized processor. For example, custom datapaths can only extend the execution pipeline stage, as shown in figure 1.3. This restricts I/O operations to the data fetched from the register file in the decode stage, severely limiting throughput of the custom instructions. Also, this approach does not look for opportunities for resource sharing between the integrated and the custom execution pathways, leading to increased resource usage.

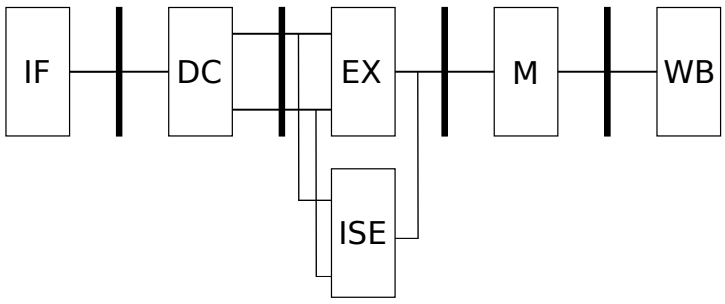


Figure 1.3: RISC pipeline with Instruction Set Extension (ISE), as found in the Nios II

We wish to take advantage of these opportunities, and generate a single datapath combining all instructions. Each instruction will give us one pattern, represented as a directed graph. Given a set of these data flow graphs and given that their execution is mutually exclusive in time, our

problem is then to generate a synthesizable description of a hardware datapath that uses as little resources as possible (surface or, for FPGAs, resource occupation rate). We will focus our efforts on datapath optimization and leave aside synthesis of control logic (instruction decoding, hazard management) for the time being.

1.2 High-Level and Multi-Mode Synthesis

Most hardware designs are still written at Register Transfer Level (RTL). These hardware descriptions can be efficiently translated, or *synthesized*, for either FPGA or ASIC. With only the more abstract data flow descriptions in hand, we will need to use more advanced synthesis techniques developed in the context of high-level and multi-mode synthesis.

1.2.1 High Level Synthesis

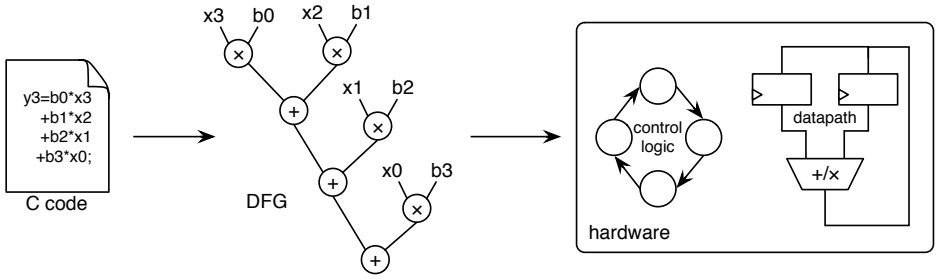


Figure 1.4: High Level Synthesis

High Level Synthesis (HLS) is the process of generating a hardware implementation capable of executing a computation that is only semantically described, generally as a Data Flow Graph (DFG). Many compilers use Data Flow Graphs as an intermediary representation for optimization purposes; high level synthesis can therefore also be applied to C programs, as long as they contain no loops of indeterminate length (loops with fixed bounds are unrolled). As these DFGs represent an entire program, they contain tens or even hundreds of nodes. For example, a common 256-tap FIR filter results in a graph with over 512 nodes. In this context, temporal resource sharing is the most important optimization process. Temporal resource sharing works by re-using the same functional unit to execute another operation in the DFG after the first is finished.

In our case, however, the DFGs representing instructions are very small, with the number of nodes rarely reaching the double digits. Also, the pipelined nature of the processor makes temporal sharing impossible, as the resources must stay available for the next instruction, which might use the same datapath. If we were to use common HLS tools to synthesize the architecture description directly, we would therefore get a very inefficient design, with a separate datapath for each instruction and a huge multiplexer to select the correct output.

1.2.2 Multi-Mode Synthesis

Multi-mode synthesis is a form of HLS that seeks to implement datapaths that can execute several algorithms. The multi-mode problem appears very similar to our problem: given a set

of DFGs, find the optimal datapath that can execute any of the individual computations.

A common example of multi-mode synthesis is the design of a mobile communication chip that can perform filtering for either EDGE, 3G, or WiFi (but only one at a time). Especially if the algorithms are similar, a lot of ressources can be shared between the different designs, leading to an important overall surface gain (compared to implementing each separately, this can be 40-80%). Given two or three algorithms in the form of DFGs, a multi-mode synthesis tool will look for both inter-design and temporal resource sharing opportunities. Some tools, such as SPACT[4], will even convert inter-design exclusion into temporal exclusion. Unfortunately, as with HLS, both the fact that no temporal resource sharing can be used and that we have a lot of very small graphs instead of a few big ones makes these algorithms ill suited for our purposes.

In fact, our data flow graphs correspond almost exactly to the datapath needed for their execution, except that the pipeline registers are not included. (Most operators have sufficient laxity in their timing that they could be placed in any of several pipeline stages, and fixing a stage would remove many opportunities for sharing, so this is not a desirable solution.) A more promising approach, then, could be datapath merging, which operates on the actual hardware operators and interconnects that implement a DFG.

1.2.3 Datapath Merging

An optimal solution to the datapath merging problem was described by Moreano et al.[11] in *Efficient Datapath Merging for Partially Reconfigurable Architectures*.

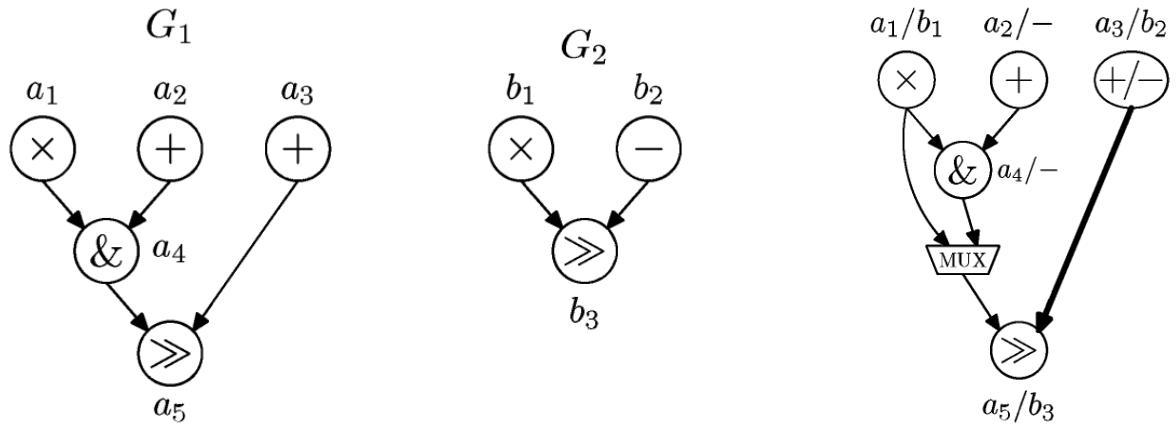


Figure 1.5: Original and merged datapath.

In their work, a datapath is represented by a graph where each node represents a functional unit and each edge an interconnect between two functional units. (These graphs are also called data flow graphs, but to avoid confusion they shall be called "datapath graph" here.)

The datapath merge algorithm will try to match nodes with similar or identical operations and merge the set of nodes that will reduce the overall surface the most. An important feature of this approach is that surface overhead due to the insertion of multiplexers is taken into account by considering edge merging in addition to node merging. Compatible or mutually exclusive matches are represented via a *compatibility graph* containing all matches, from which the optimal set of matches is extracted.

This algorithm is especially well suited to our many small graphs, as fewer nodes mean a smaller compatibility graph and thus a lower execution time. However, our problem has two additional aspects that influence which matches are feasible: operators can have mobility across pipeline stages, and critical path length between two pipeline registers is strongly constrained by the clock cycle length.

1.2.4 Proposed Approach

Our approach is based on Moreano's algorithm. We propose to modify the datapath merge algorithm in order to take into account the above constraints while constructing the compatibility graph. Merges are only declared compatible if they will not stretch the critical path beyond the clock cycle length, and if their timing is not mutually exclusive.

In the following chapter, we will present an in-depth analysis of existing work, which will be followed by a detailed explanation of our approach in chapter 3.

Chapter 2

Existing Approaches

This chapter details existing approaches that implement resource sharing in different contexts. The first two are classical Data Flow Graph merge algorithms for multi-mode architectures. The third extracts similar computation patterns from one or several algorithms, making it possible to detect opportunities for resource sharing already during the ISE extraction. The last two approaches work at a lower level, modifying the datapaths implementing one or more algorithms, but from two diametrically opposed directions: while trimming consists in first constructing the most complete architecture possible, then progressively removing little-used components, datapath merging iteratively adds each functionality while trying to use as few resources as possible.

2.1 Adapting High-Level Synthesis Algorithms

An early approach to the multi-mode problem was to use existing high-level synthesis algorithms by presenting the problem in a way that would expose the opportunities for resource sharing to the tool. The authors of [4] observe that current scheduling and binding algorithms are quite intelligent, and that resource sharing can therefore be obtained by first scheduling all DFGs separately, then concatenating them using dummy nodes before handing them over to the binding algorithm. This will introduce a fake dependency between the nodes, leading the synthesis tool to declare them mutually exclusive in time.

However, the test set they used to evaluate the performance of this approach is insufficient, being comprised of only two sets of DFGs: one containing four FIR filters with different taps, the other one FIR filter and two 4th-order IIR filters.

If we suppose that the results obtained with this set are representative, the algorithm does not perform too badly: delay overhead is 7% on average (similar to the other approaches [2, 3, 6]), and area reduction is a reasonable 45% (the other approaches yield between 50-70%, according to their respective authors).

Any results will only be as good as the synthesis tool used to perform the scheduling and binding. For this reason, Chavet et al.[3] and later Le Gal and Casseau [6] developed dedicated scheduling and binding algorithms for this approach. However they are mostly concerned with register allocation and register file management, which are not relevant in our case (these components are fixed in the processor description).

2.2 Longest common substring

This approach, presented by P. Brisk, A. Kaplan, and M. Sarrafzadeh in [2] implements a combined scheduling and binding algorithm. This allows them to make significant tradeoffs in latency in order to gain more surface.

In fact, it makes no latency promises at all: the authors suggest to run some latency-constrained resource-estimation algorithm to determine the number of functional units needed, and then to check afterwards if the design respects the timing constraints.

This algorithm does not work iteratively, but merges all graphs at once. It is a polynomial-time heuristic based on the search for the longest common substring between all paths in the graphs: for this, all graphs are first decomposed into sets of paths (cf. figure 2.1(a)). These paths are then searched for the common (between at least two graphs) substring with the largest area (b), which is merged first (c). All the graphs exhibiting this maximal substring are then merged by iteratively finding the largest substring among the not-yet-shared vertices (d,e,f). Once this is done, the search for the longest common substring between all graphs is started again, until only one graph remains.

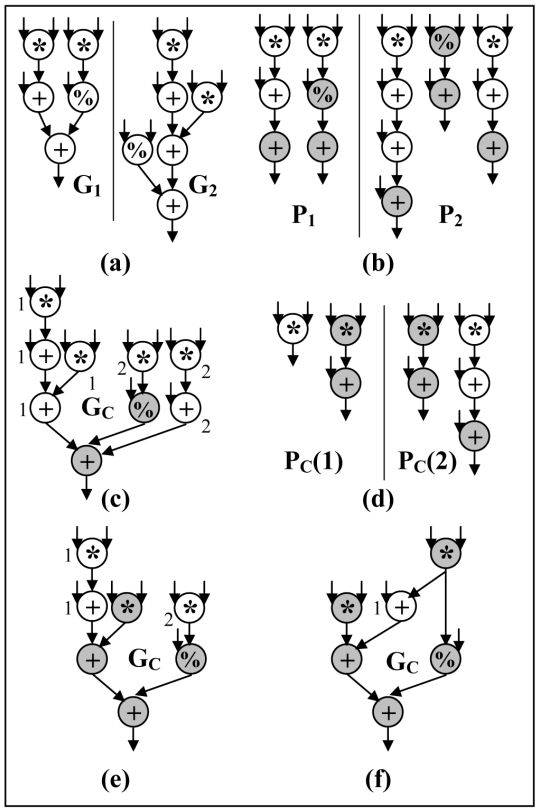


Figure 2.1: Execution of the longest common substring algorithm on two example graphs. It is supposed that $area(\%) > area(*) + area(+)$.

When the merged graph has been calculated, interconnects are established. This step takes into account commutativity of operators to minimize the number of inputs on each multiplexor.

This approach seems to imply that the authors consider it always more efficient to share even a single node, at the cost of inserting up to two multiplexors, than not to share it. This is a not a reasonable supposition, as some functions (especially logical bitwise operations) can be significantly smaller than a multiplexor. On FPGA this effect will be even more marked than on ASICs, as multiplexors are so very expensive: a simple 4:1 multiplexer will generally use twice as many look-up-tables as a bitwise logical operation.

2.3 Pattern detection

In [5], Jason Cong and Wei Jiang present an approach that does not merge graphs, but rather detects common patterns among them. Unlike the pattern detection commonly used for ISE extraction, they also detect similar but not necessarily identical instances and count them as the same pattern up to a certain distance. The measurement of this distance, called *editing distance*, is defined as the cost of the cost-minimal sequence of edit operations that transforms

one graph into the other.

This takes into account some very different resemblances between nodes than the other approaches, such as variations in bit-width (a 32-bit multiply-accumulate pattern can be used to calculate a 16-bit FIR filter with very little modification). It can also capture similarities in structure such as subtraction/comparison.

To find patterns, the authors start by iteratively enumerating patterns of increasing size using a breadth-first search. Starting with all collected patterns of size k, patterns of size $k+1$ are constructed by enlarging patterns of size k with adjacent nodes. The number of instances of each pattern is then counted, and only patterns with more instances than a threshold value l_{count} are kept for the next step.

To make it easier to estimate the combined latency of a pattern, only *convex* patterns are considered. A pattern is convex if none of its instances has a path between two nodes of the pattern that includes a node that is not part of the pattern.

Finding all instances of a pattern is yet another instance of the *subgraph isomorphism problem* and hence NP-complete. To accelerate the average execution, the authors use two heuristics: first, the *characteristic vector* of a subgraph can be used to determine if it is too far different from the pattern to possibly be an instance, reducing the number of graph comparisons needed. Second, *locality sensitive hashing* is used as a probabilistic way to find similar graphs: a locality-sensitive hash function hashes similar graphs to the same value with high probability, and has a high probability of hashing dissimilar graphs to different values.

Sometimes, culling patterns with too few instances is not enough to keep the number of patterns down to a manageable level. If an application contains very large patterns, many patterns will be found since every subgraph of a pattern is also a pattern. If the number of patterns exceeds a certain threshold, the pattern recognition tool will switch to a depth-first exploration of the graphs in order to find maximal patterns.

Once all patterns have been enumerated, the best patterns are selected for implementation. The authors propose the following measure for goodness, where P is the considered pattern and N the number of times it is found in the graph:

$$\frac{N * mux(io) + area(P)}{N * (mux(io) + mux(internal)) + area(P)} + \alpha * \frac{|P|}{latency(P)}$$

The first part of this equation describes the area saving of using this pattern, and the second part prefers patterns exhibiting more parallelism. Parameter α can be adjusted to influence the amount of latency overhead tolerated.

Finally the selected patterns are scheduled using any classical scheduling algorithm that can take into account relative timing constraints (these are used to make sure that all instances of a pattern are scheduled the same way).

This is a very interesting approach, but it is outside of the scope of our work, as we start with a finished ISE. However, it would be interesting to test whether the improvements to resource utilization introduced at the ISE extraction stage are cumulative with our gains or whether they cancel each other out.

2.4 Trimming

This approach, described in *Automatic Architecture Refinement Techniques for Customizing Processing Elements* by Bita Gorjiara and Daniel Gajski[7] is aimed at No-Instruction-Set-Computer (NISC) architectures. In NISC architectures, instead of controlling the processor's operations via instruction words, the underlying control architecture is directly bared, and the different parts of the compiled (nano-)code directly control the functional units and interconnects.

Here, the authors start with a complex general-purpose architecture, and iteratively remove un- or underutilised components. They analyze the nanocode generated by the compiler to determine which values are used for each control word. Based on this, they can determine utilisation of:

Registers: these can be removed if they are never used in the program.

Multiplexers: these can be removed if unused, or resized if some values are never taken.

Multi-operation units: some of the operations of an MPU may never be used, in which case the corresponding component can be removed.

Register files: if the maximum number of simultaneous reads or writes allowed is never reached, the number of ports in the register file can be reduced. The number of registers in the file can also be optimised by analysing the maximum register requirements of the program.

Constant fields: if the largest constant in the program uses less than the available bit-width, the size of the constant field can be adjusted.

As many of these optimisations modify the control interface, the nanocode is modified after each step to reflect the introduced changes.

The code is then either simulated or profiled to determine utilisation rates of each remaining component. Based on a threshold value, the trimmer determines underused resources that can be merged into other components capable of performing the same function. After each merger the program is recompiled in order to detect unnecessary connections introduced during merging. The more complete the original architecture, the better the final result of this procedure.

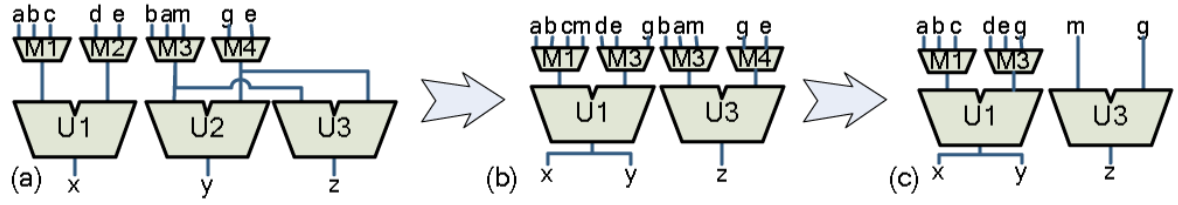


Figure 2.2: Trimming of an underused ALU and subsequent input reordering via recompilation.

The trimming algorithm could be modified to stop removing components when timing constraints intervene instead of stopping when utilisation is sufficiently high, but this approach would probably not translate well to our problem: the fixed instruction set we start with offers far less flexibility (and hence occasions for optimisation) than nanocodes, and the architecture is less complete.

2.5 Datapath Merging

Datapath merging is the opposite to the previous approach, as instead of starting with a large design and removing as many resources as possible, it starts with a small design and tries to add as few resources as possible.

The datapath merge algorithm described by Moreano et al.[11] works by finding all possible *mappings* between nodes and between edges from two datapath graphs, and noting the surface that can be gained by merging the two. It also notes whether mappings can be applied simultaneously or whether they are mutually exclusive. From this it derives the set of compatible mappings that maximise the cumulative surface gain, resulting in a merged graph that represents the optimal datapath fulfilling the function of both.

2.5.1 Datapath Graph

First, let us define exactly what a datapath graph is. A datapath graph is a representation of a hardware configuration, where a node represents an operator or functional unit (adder, multiplier, register, multiplexer...) and an edge represents an interconnection (wire). We will illustrate the merging algorithm with the two datapath graphs in figure 2.3.

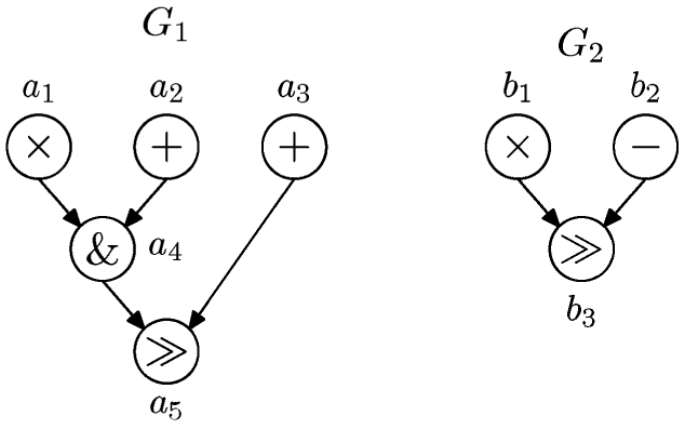


Figure 2.3: Two datapath graphs

2.5.2 Mapping

The first step of the algorithm is to identify possible mappings. A mapping is a correspondence between two components whose function can be performed by a single element. There are two kinds of mappings: node mappings and edge mappings. Each mapping has a weight representing the surface area that can be gained by merging the concerned components.

A node mapping is a correspondence between two nodes representing similar or identical operations. For instance, two adders can be mapped, or an adder and a substractor can be mapped if there exists a library component capable of executing both operations. The weight of a node mapping is the difference between the size of both operators separately and the size of the combined operator together with the multiplexers needed to choose between the different inputs.

$$W_{node\ mapping} = W_{node1} + W_{node2} - W_{merged\ node} - W_{mux}$$

An edge mapping is possible when two adjacent nodes are merged. In this case, there is one less input for the second node, thus only a smaller or even no multiplexer is needed. Its weight is therefore the area gained by reducing or removing the multiplexer.

$$W_{edge\ mapping} = W_{mux}$$

For our two example graphs, the mappings of figure 2.4 are possible.

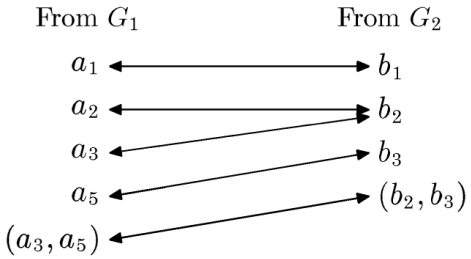


Figure 2.4: Possible mappings between G_1 and G_2

2.5.3 Compatibility Graph

Two mappings can be incompatible, i.e. it is impossible to implement both at the same time. This happens when a node has several matches in the second graph; only one of them can be chosen. The way this is represented is by a weighted *compatibility graph*, where each node represents a possible mapping, and all compatible mappings are connected by an edge.

Once this compatibility graph is constructed, finding the set of compatible mappings that maximise the cumulative surface gain becomes equivalent to finding the set of nodes that are all interconnected that have the greatest cumulative weight, called the *maximum weight clique*.

Figure 2.5 illustrates the compatibility graph constructed for our example, with the maximum clique highlighted in bold.

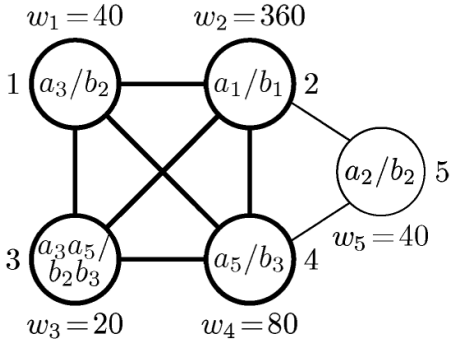


Figure 2.5: Compatibility graph

2.5.4 Maximum Weight Clique

Identifying the maximum weight clique in a graph is an NP-complete problem. Integer Linear Programming can be used to solve this problem, but to avoid very long execution times it is generally preferable to use a heuristic. Heuristic algorithms cannot guarantee that they will find the optimal solution, however a good heuristic will deliver a result that is good enough so that it will not be worth waiting for the small additional benefit the optimal solution might bring.

In our case, as the DFGs are small, with few possible matches, we will generally have simple compatibility graphs. Therefore, even the more basic heuristics should be able to find the optimal solution in most cases.

2.5.5 Merged Graph

The maximum weight clique indicates which nodes and edges need to be united in order to construct the merged datapath graph. This datapath is able to fulfil the role of either of the original datapaths, however due to the additional multiplexers its critical path may be longer than the maximum of both original critical paths.

Figure 2.6 shows the result of the datapath merge algorithm on our example graphs.

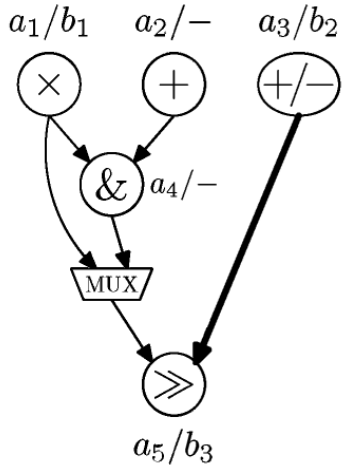


Figure 2.6: Merged datapath

This algorithm is the closest to our problem among the approaches we studied, we have therefore chosen to use this algorithm and adapt it to our problem. The following chapter will explain how we achieve this.

Chapter 3

Proposed Modifications

This chapter will show how we propose to modify the datapath merge algorithm by Moreano et al. to adapt it to our problem. There are three areas of improvement that we worked on:

1. In order to make the datapath merge algorithm aware of the pipeline's timing constraints, we propose to annotate all operators with the earliest and latest time where their execution needs to start (ASAP/ALAP). By placing additional conditions on which nodes can be merged, we can then ensure that these timings are respected in the merged graph.
2. The Moreano merge algorithm can lead to combinatorial loops, which render the design unsynthesizable. We check for possible loop formation and prevent it by declaring the involved mappings incompatible.
3. On FPGAs, the resources needed to implement a multiplexer can vary drastically depending on the environment. As an example, when implementing an adder, it is in most cases possible to multiplex with a third input at no extra cost, but if a fourth is introduced, an extra logic bloc is needed that is as large as the adder itself. We will therefore also present the model used to calculate multiplexer weight in our implementation.
4. A further opportunity for resource sharing exists: some operators can be broken up into smaller operators. We propose two ways to integrate this into the datapath merge algorithm

The following sections will detail each of the proposed modifications.

3.1 Timing Conditions on Mappings and Compatibility

3.1.1 Constraints

In our problem, there are two kinds of timing constraints to take into account: explicit constraints from the PSTK description, and pipeline constraints.

Explicit constraints occur for memory and register accesses. Each register and memory bank has a specified pipeline stage at which it can be read and another at which it can be written. It must therefore be ensured that no access happens outside this specified time frame.

The pipeline imposes additional constraints on operator executions. Contrary to the hypotheses needed for the standard datapath merge algorithm, the different instructions are not totally mutually exclusive time-wise, only by pipeline stage. For instance, let us consider the graphs in figure 2.3 (p.14), and suppose for simplicity's sake that all operators have the same latency, and that one clock cycle is long enough to accomodate one operator. Let us further suppose that the result of G_1 needs to be available after three clock cycles, and the result of G_2 after two. With the pipeline registers inserted, this gives us the graph in figure 3.1. In a classical multi-mode case, where one of the modes is chosen and the corresponding calculation performed a large number of times before switching modes and exclusively performing the other computation, both modes can have different latency requirements. It is only necessary to verify that the length of any path between two pipeline registers is always inferior to the length of one clock cycle. (In this case, that means verifying that the combined latency of a logical AND and a multiplexer is shorter than a clock cycle.)

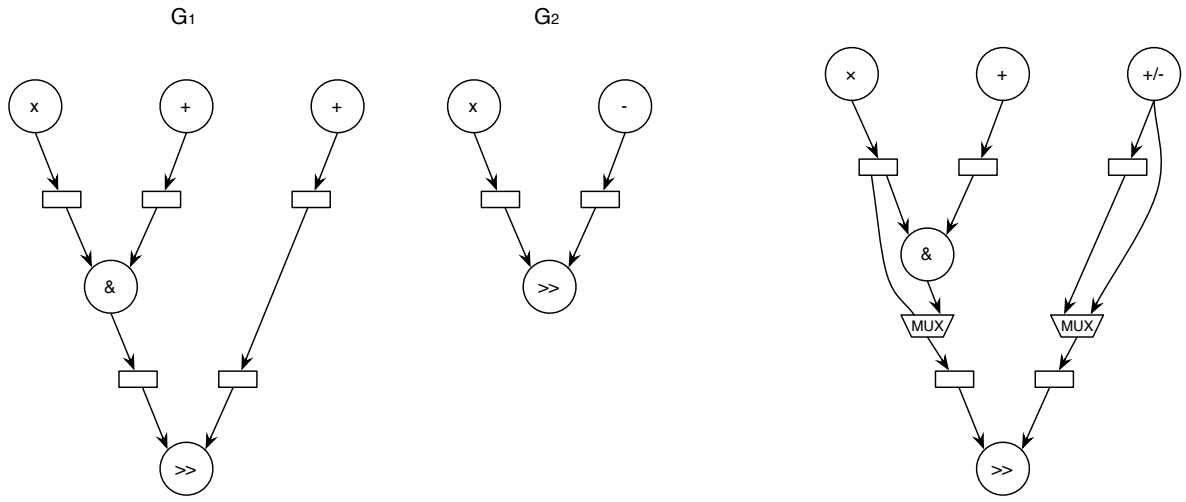


Figure 3.1: Pipelined datapaths and their fusion.

However, if as in our case the different pipeline stages are executing different computations, this kind of merging leads to problems. For instance, if an instruction specifying the calculation performed by G_2 follows immediately upon an instruction specifying G_1, G_1 will be in its third stage, performing a right shift, when G_2 is in its second stage, also asking for a right shift. Two separate functional units that can execute a right shift are therefore needed. In addition to verifying that the insertion of multiplexers will not cause any timing violations, it is therefore necessary that no operators whose timing is mutually exclusive are merged.

If we modify one of the hypotheses, and assume that the result of the right shift operation in G_2 is not actually needed at the end of the second clock cycle, but only at the end of the third, a merge would be possible, leading to the merged graph of figure 3.2.

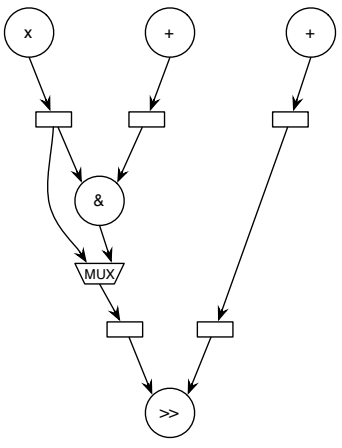


Figure 3.2: Merged graph for lesser latency requirements

3.1.2 Constraint Enforcement

The ASAP and ALAP scheduling that is performed on the individual graphs before the merging takes into account the prescribed stage for register and memory accesses.

A mapping is authorized when the intervals between ASAP and ALAP scheduling of both nodes intersect. The resultant node will then be scheduled during this intersection.

This local condition by itself is not enough to ensure that timing is respected. In situations where two dependent nodes with great laxity are mapped with nodes with mutually exclusive domains, timing violations can occur. It is therefore necessary to detect that these mappings are incompatible and to remove the concerned edge in the compatibility graph.

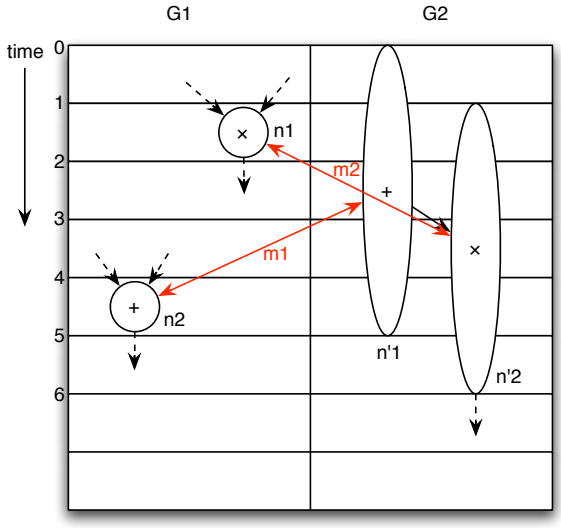


Figure 3.3: Two admissible mappings that are mutually incompatible

As an example, let us consider two graphs containing the nodes represented in figure 3.3. The admissible timing interval [ASAP; ALAP] of each node is represented by the space it covers vertically. n_1 can be merged with n'_2 and n_2 with n'_1 as they each have possible timings in common. However, if both mappings were merged at the same time, the calculation in G_2 could not be performed, as the multiplication n'_2 would have to take place during the interval $[1; 2]$ despite the fact that one of its operands is the result of the addition n'_1 which can only be performed during $[4; 5]$. This violates causality.

When checking whether two mappings are compatible, we will therefore implement the following tests:

- If for both graphs the nodes of this graph belonging to each mapping are not connected, they are compatible.
- If there is a connection (possibly spanning several nodes), the ASAP and ALAP times after merging are calculated and it is checked that each operator between the two mappings has enough time to be executed.

For each path p connecting a node of the first mapping m_1 to a node of the second mapping m_2 (the order of the mappings is determined by the direction of the path p), the following needs to be verified:

$$ALAP(m_2) > ASAP(m_1) + length(p)$$

3.2 Combinatorial Loop Detection

The original datapath merge algorithm does not prevent combinatorial loops from occurring. A combinatorial loop is a circuit where an output feeds back into an input with no register interposed. If a loop occurs, the critical path will become infinite, and the design cannot be synthesized.

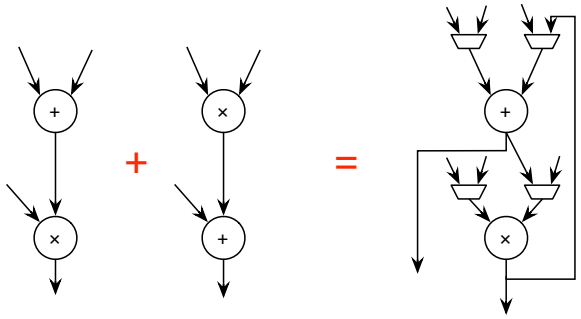


Figure 3.4: Combinatorial loop occurring in a merging

The loops generated in the datapath merge algorithm are not "true" combinatorial loops, because the multiplexers inserted into the design should always be configured in a way that prevents the loop from occurring. However, synthesis tools will not be able to determine that this is the case (the boolean satisfiability analysis required is NP), and refuse to synthesize the design.

In order to prevent this, mappings that could lead to cycles are declared incompatible. This is the case when there exists a path from a node of the first mapping to a node of the second mapping in one graph, and a path from a node of the second mapping to a node from the first mapping in the other.

3.3 Multiplexer Weight Calculation

Due to the structure of FPGAs, multiplexer implementation is far more complex than on ASIC (where it is a straightforward component whose size grows linearly with the number of inputs). An FPGA is composed of reconfigurable logic blocks called Look-Up-Tables (LUT) that can emulate any boolean function of 3-6 variables (depending on the model and configuration). For instance, if we have an FPGA composed of 4-input LUTs, and use it to implement a bitwise and, only two inputs of each LUT will be used. We could therefore use the remaining two inputs for a multiplexer, one for the additional operand and one for the selection command.

We will use Altera's Stratix IV devices here to show how to calculate appropriate multiplexer weights for such an architecture. For other devices with different structures the calculations

would have to be modified, but the general idea remains the same.

The Stratix IV has special modes for adders, substractors, multipliers, left/right and barrel shifters. Multipliers and shifters are allocated to special DSP blocks, and support only two inputs. Logic blocks in add/sub mode can multiplex one additional input.

All other functions are implemented in "normal mode". Normal mode combines two LUTs into one fracturable logic block with eight inputs and two outputs. The LUTs can divide the inputs in different ways: two 4-LUTs, a 5-LUT and a 3-LUT, two 5-LUTs with two shared inputs, etc. An overview of the different configurations can be found in figure 3.5.

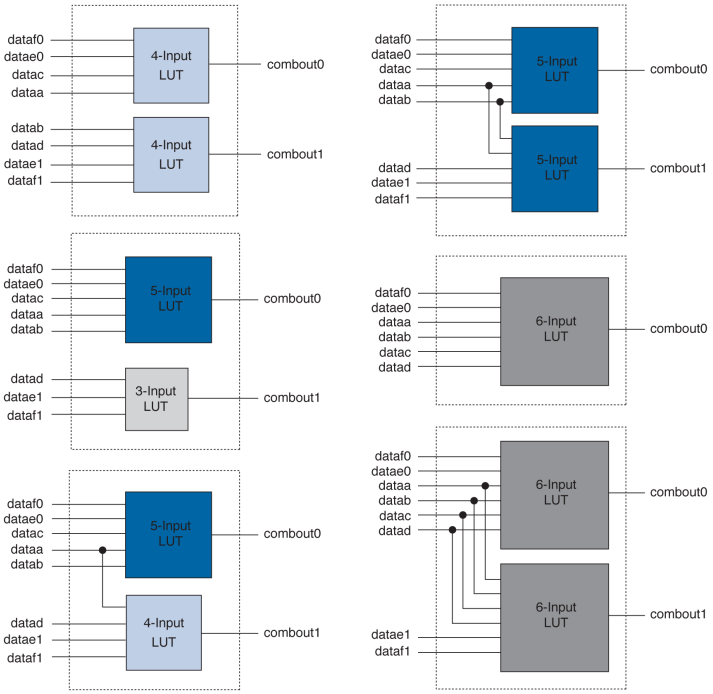


Figure 3.5: Possible LUT configurations in normal mode

To simplify, we will measure size in "half-blocks", and suppose that a half-block can hold either a 3-LUT, a 4-LUT or a 5-LUT. This should even out overall, as two 4-LUTs or a 3-LUT and a 5-LUT can be packed together into one logic block. Size estimations are not as precise in practice anyway, as the resources used by a specific design are strongly context-dependent: the synthesis tool can pack parts of different operators into a same block, and will sometimes use more than the optimal number of LUTs in order to ameliorate timing.

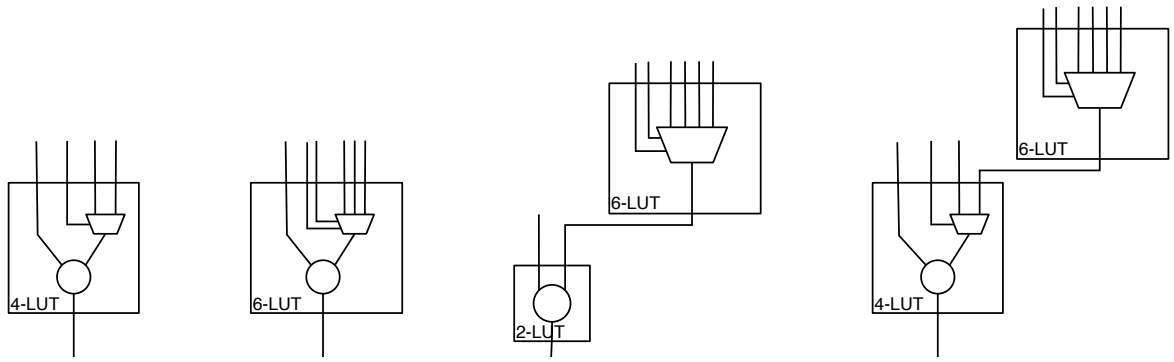
We will also suppose that it does not matter how the inputs are distributed on the input ports. Thanks to the flexibility of the fracturable logic blocks, the resources used are approximately the same. We verified this empirically up to 8 inputs, a number that should seldom be exceeded in our datapaths.

Our multiplexer cost estimation, then, is as follows: for a binary operator with x inputs, $(x - 2) - \left\lfloor \frac{x}{3} \right\rfloor$ half-blocks are needed to implement multiplexers.

As an example, we will show what happens when all additional inputs are multiplexed for one port. For a bitwise binary operator, one additional input can be multiplexed within the original half-block, as shown in figure 3.6a. If we add another input, together with the selector bits we will have 6 inputs total, which means that the operator and the multiplexer will fit into one

logic block in 6-LUT mode (fig. 3.6b). Beyond that, multiplexers have to be implemented in separate LUTs. A 2:1 mux fits into a half-block and a logic block is designed to accomodate one 4:1 multiplexer[13, 8], which means that the multiplexer cost will be increased by a half-block every 3n+1 and 3n+2 inputs. Every third input is free, as it will fit into a underused half-block: as per our hypotheses, the 2-LUT in figure 3.6c and the 4-LUT in figure 3.6d take approximately the same space. This pattern of growth can be repeated indefinitely by treating the output of the 6-LUT as one input and proceeding with the 4-LUT as from the beginning.

We plan to check this resource occupancy estimation against the output of the Quartus II synthesis tool to see if these packing rates can be obtained in practice.



(a) 3 inputs (b) 4 inputs (c) 5 inputs (d) 6 inputs

Figure 3.6: LUT growth with inputs.

3.4 Operator Decomposition

The more complex operators, such as multipliers, can be divided into several simpler blocks (cf. fig. 3.7). This can lead to partial correspondence between operators: e.g., a 32-bit multiplier can be merged with up to four 16-bit multipliers. The bitwidth and other characteristics are specific to the targeted FPGA model, so information about the target needs to be provided during graph construction.

There are two ways of integrating this kind of resource sharing. A simple solution is to construct each graph with the operators already cut up into several nodes. This requires no modification of the algorithm at all, but it means that the decomposition is static. Sometimes, it might be desirable to divide an operator differently depending on the node with which it is mapped.

In that case, mappings which use the same

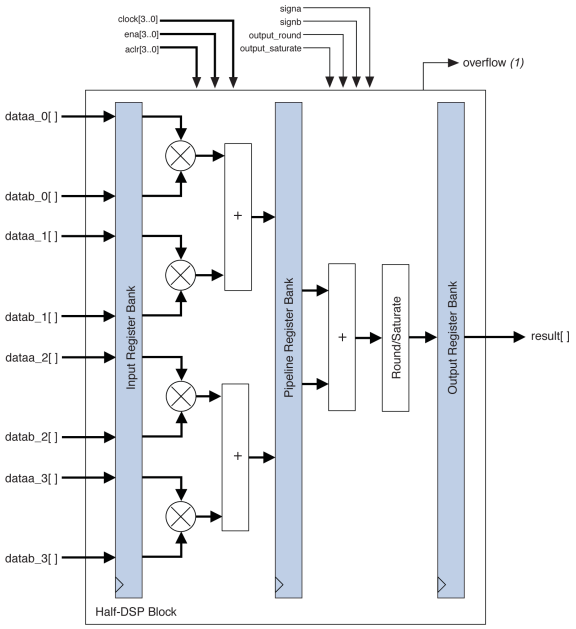


Figure 3.7: Structure of a multiplier

decomposition but separate parts of the operator would be compatible, whereas mappings which use different decompositions would have to be marked incompatible.

Operator decomposition can increase the size of the compatibility graph dramatically, rendering the maximum weight clique intractable, especially in cases with a lot of internal similarity. We have not yet found a way to successfully merge a decomposed multiplier.

Chapter 4

Implementation

This chapter will detail how we realized the modifications detailed above, based on a pre-existing implementation of the datapath merge algorithm.

In addition to datapath merging, the whole transformation chain from processor description (text file) to VHDL code was implemented using model-driven engineering. Several existing meta-models served as the basis for this, which shall be briefly presented in the following.

4.1 Meta-Models

4.1.1 PSTK

PSTK is a domain-specific language for describing a processor instruction set and memory layout developed by a M1 project group using the development framework *Xtext*. Xtext is an Eclipse plugin in the Eclipse Modeling Framework aimed at developing domain specific languages (DSLs). The user describes the grammar of the DSL, and Xtext will generate a parser (using Antlr), an Eclipse text editor for development in the DSL, and an Ecore model. The grammar-based processor models generated by PSTK therefore take the form of abstract syntax trees enhanced with symbol linking information. These cannot be worked on directly, we therefore extract the necessary information to construct the working set of DFGs.

4.1.2 The `IGraph` model

The CAIRN team has developed a small library of graph analysis and manipulation algorithms and tools that were used extensively during this internship. In particular, this library contains the implementation of the DFG merge algorithm of Moreano et al. that we adapted. All these algorithms work upon instances of the `IGraph` model, our DFGs therefore implement the interfaces specified in it. An `IGraph` consists of `INodes` containing `IPorts` that are connected by `IEdges`.

4.1.3 The `Datapath` model

`Datapath` is a model developed by the CAIRN team whose main feature is automatic translation to VHDL code that can then be synthesized for FPGA. As such, the description is close to hardware concepts, based on logic blocks and storage elements interconnected by wires. In VHDL, the basic building blocks are *components*, which have labeled inputs and outputs that can be connected by wires. As such, the elements of the `Datapath` model can each generate their corresponding VHDL code without needing any further information about the environment.

4.2 The Generation Workflow

The generation workflow is similar to a classical compilation workflow: the processor description code is parsed and an AST is constructed; this AST is then traversed to construct an intermediary representation in the form of data flow graphs, on which various optimization routines are performed; the resulting graph is then transformed into a datapath representation from which VHDL code can be generated.

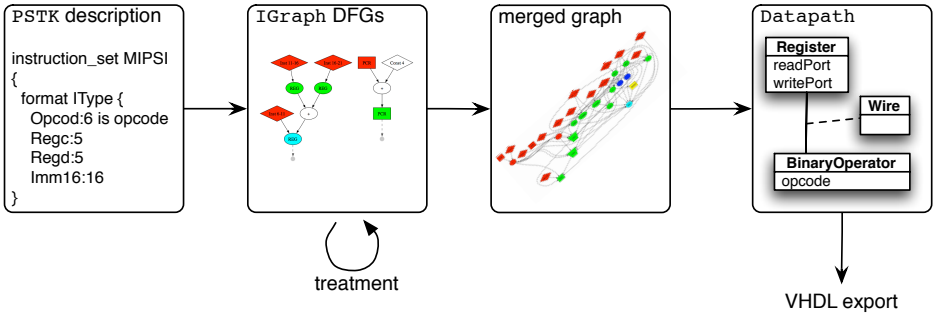


Figure 4.1: The generation workflow

4.2.1 Processor Description to Instruction DFG Translation

Based on the PSTK grammar, the Xtext eclipse plugin generates a parser/linker that reads the given processor description, generates an AST and performs symbol linking. This linked AST is then handed over to the translator class that handles DFG extraction.

The AST is traversed, and for each instruction a graph is constructed, containing two unconnected parts: the computation flow and the program flow. The computation flow may be empty or contain several unconnected parts. The program flow always consists of a tree writing the program counter at its root.

The DFGs implement the `IGraph` interface. During graph construction, the `INodes` are annotated with several elements. All nodes contain at least typing information, which is necessary to compare nodes. Nodes representing memory or register accesses also contain information about the pipeline stage assigned in the code. A representation of a resulting DFG can be found in figure 4.2, with the colors standing for the different pipeline stages.

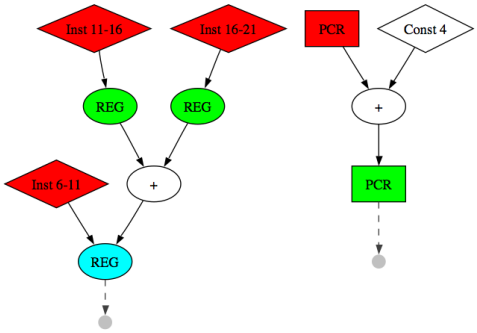


Figure 4.2: The DFG resulting of the translation of the ADD instruction.

4.2.2 Individual graph scheduling (ASAP/ALAP)

Each individual graph is then scheduled, taking into account the constraints on register and memory accesses.

A pre-existing ASAP and ALAP scheduling algorithm was modified. When a register or memory access is encountered, the corresponding interval of allowable computation start times is restricted to the time of the specified pipeline stage.

Additionally, the interval of time that is closer to the end of a pipeline stage than it would take to execute the operation is removed from the interval of possible schedule times.

4.2.3 Common subexpression elimination

Next, each instruction DFG is traversed to detect common subexpressions and remove unnecessary double calculations. In the example in figure 4.3, $PCR + 4$ is calculated twice in the original graph. After common subexpression elimination, both nodes using $PCR + 4$ are fed from the same output.

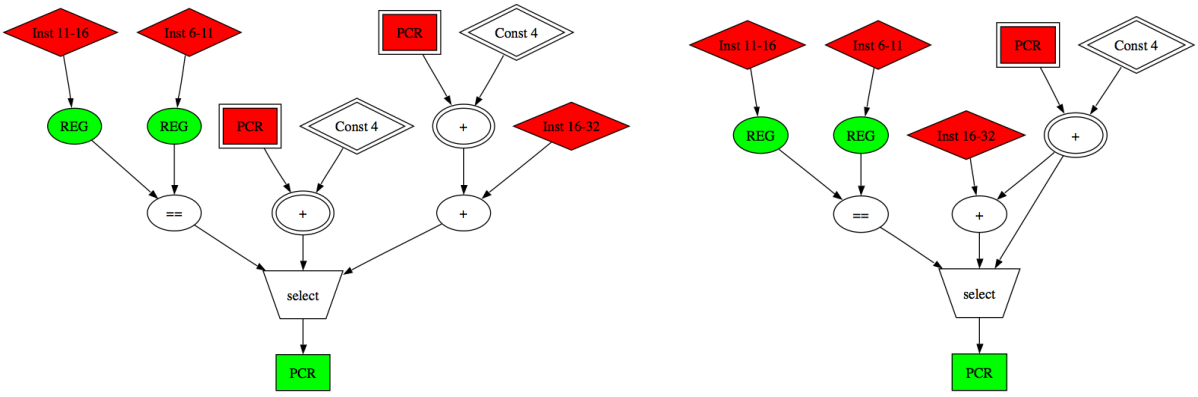


Figure 4.3: Common subexpression elimination for a conditional jump instruction.

It is not necessary to check timing for this step, as common subtrees will always start at the leaves and thus have the same ASAP timing. We perform timing verification anyway, as a sanity check.

4.2.4 Graph merging

The instruction DFGs are merged using a modification of the library graph merging algorithm. On top of the features described in the algorithm specification in chapter 2.5, the pre-existing implementation also prevents combinatorial cycles from being formed.

The modifications to mapping and compatibility detection detailed in chapter 3 were implemented, except for prevention of combinatorial loops, which was already implemented by a previous intern at CAIRN. Operator decomposition was removed again because it resulted in too large compatibility graphs.

Thanks to a code structure extensively based on the design pattern `Provider`, the core algorithm code (compatibility graph construction, max clique heuristic and merged graph construction) did not need to be modified. Only the match and compatibility detecting methods needed to be changed.

A slight problem with the `IGraph` model during this step is that it is purely topological, and contains no semantic information. While the nodes are annotated with type and other information, this is not used by algorithms using the `IGraph` abstraction. This becomes a problem for port matching, as ports are only identified by their place in an `INode`'s list of ports. As the order in which ports are created is dependent of the AST, it is not always possible to finely control it. This makes port matching troublesome for the non-commutative operators.

We plan to remedy this problem by defining a new model that will implement both `IGraph` and a semantic description of operators and their inputs.

4.2.5 From Merged Graph to Datapath

The merged graph represents a combined datapath that can execute any of the original processor description's instructions. This graph is translated into individual circuit components. The structure of the `Datapath` model is very similar to that of `IGraph`, with functional units occupying the same role as `INode`s, equipped with input and output ports and connected by `Wire`s, making the transformation straightforward.

This work is currently in progress.

4.2.6 VHDL export

An automatic exporter for instances of the `Datapath` model was developed by the CAIRN team in parallel to this internship.

Chapter 5

Experimental Results

5.1 Preliminary Results

As the the last stage of the transformation chain towards a synthesizable datapath description is still in progress, it has not yet been possible to see the actual occupation rate in an FPGA resulting from the datapaths we generate. However, the datapath graphs being very close to the final result, it is possible to see general trends in resource usage.

We described 31 standard MIPS instructions in PSTK, as well as a MAC and a discrete FFT butterfly instruction. Independently, these represent a total of 440 nodes in 33 DFGs. Our final merged graph, represented in figure 5.1, counts 47 nodes. The order in which the graphs are merged has very little influence on the resources used by the final result, but ordering the graphs in decreasing size (by number of nodes) kept the compatibility graphs slightly smaller, reducing execution time.

Without the two custom instruction, the final graph contains 32 nodes, which is the minimum amount of functional units necessary to perform all the calculations corresponding to the different instructions. The addition of the MAC instruction requires an additional adder as well as two extra register accesses (the MAC instruction has three inputs, and outputs into a dedicated register). The FFT butterfly increases the design by an additional 12 nodes, however most of the increase (9 nodes) is due to the slice selection used in the FFT butterfly to decompose input words. These do not represent a real size increase as they are implemented simply as routing and take no space. Two more of the additional nodes are due to a bug in common subexpression elimination, so the actual increase is only 1 node: a fifth adder.

The tradeoff is a significant increase in interconnection logic, i.e. multiplexers: the number of edges in the graph increases from an initial 79 to 119. If we remove the edges created by the slice selection operators, which are not present in the actual design, we are still left with 100 edges, i.e. a 25% increase in interconnections.

5.2 Future Experiments

Once the translation to `Datapath` is complete, we plan to compare the synthesis results of our merged datapath graph to both a hand-coded MIPS implementation and the result of datapath synthesis with the GAUT tool from Lab-sticc[9].

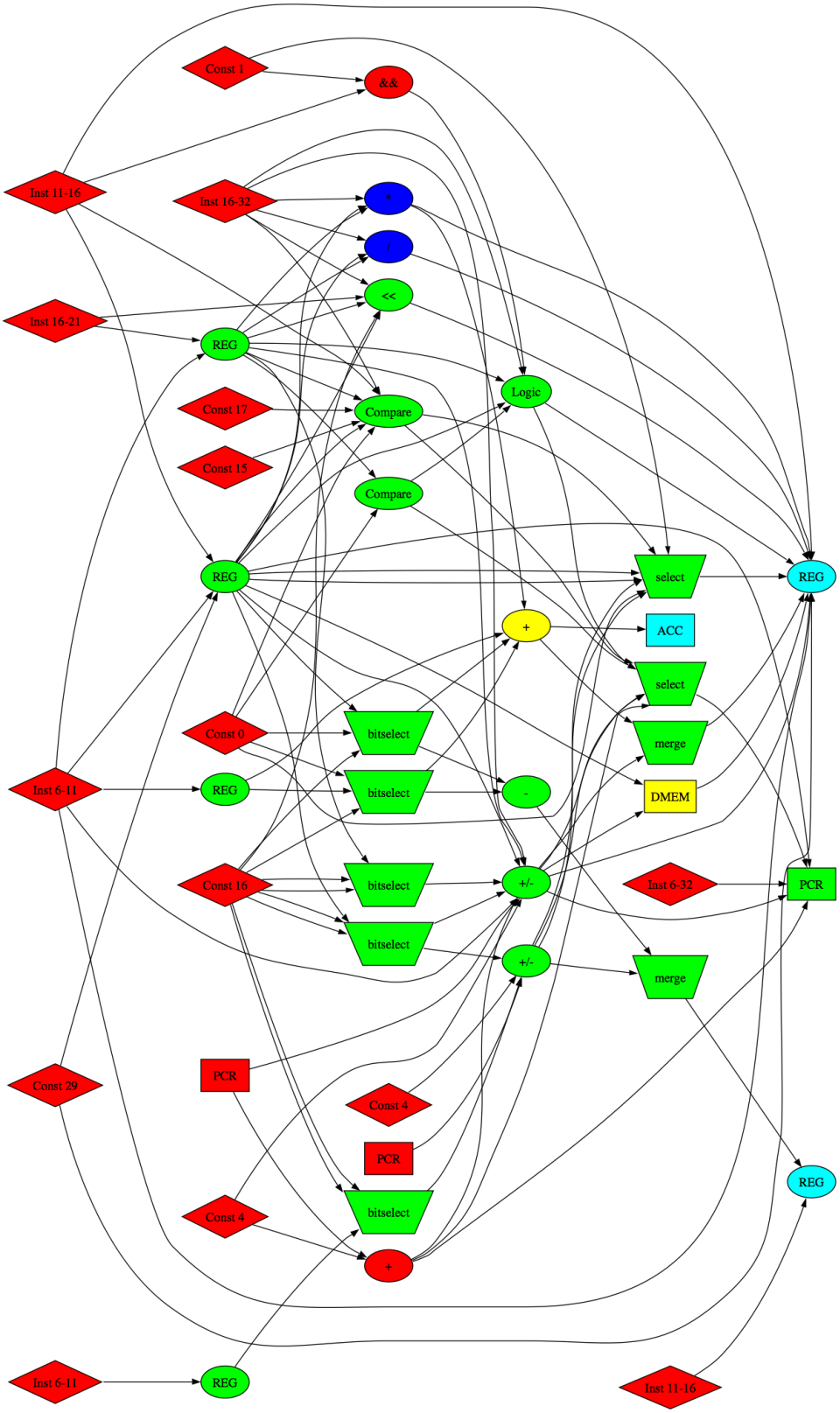


Figure 5.1: Result of the merging of 33 instructions in MIPS format.

Conclusion

Despite some focus in the recent years on the problem, automatic generation of Application-Specific Instruction-set Processors is still insufficiently resource-efficient compared to human design. The current best effort relies on extending existing processor cores with custom instruction pathways, an approach that has several drawbacks: resources already existing in the core cannot be reused, and data access is restricted to the original register reads and writes, creating a bottleneck in throughput.

The examination of techniques used in the related fields of high-level and multi-mode synthesis revealed some approaches that could be adapted to our problem. In particular, the technique of datapath merging is sufficiently close to expect that it will adapt well.

We presented and implemented an ASIP synthesis algorithm that turns a semantic description of an instruction set into a synthesizable description of a processor's datapath in an efficient manner, based on a modification of Moreano's datapath merge algorithm. We added support for operator mobility across pipeline stages, combinatorial loop prevention and realistic multiplexer resource usage estimation for FPGAs.

We also implemented the translation chain from a processor description in PSTK to the intermediary graph representation of each instruction datapath and from the merged datapath graph to a synthesizable datapath description.

Future work includes dynamic operator decomposition to augment resource sharing between similar composite operators without provoking combinatory explosion of the merge algorithm, a meta-model for the semantic aspect of the DFGs to make it easier to define matching nodes, as well as control logic generation.

Bibliography

[1] Altera Corporation. *Nios II Custom Instruction User Guide*, May 2008.

[2] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 395–400, New York, NY, USA, 2004. ACM.

[3] Cyrille Chavet, Caaliph Andriamisaina, Philippe Coussy, Emmanuel Casseau, Emmanuel Juin, Pascal Urard, and Eric Martin. A design flow dedicated to multi-mode architectures for DSP applications. In Georges G. E. Gielen, editor, *ICCAD*, pages 604–611. IEEE, 2007.

[4] Lih-Yih Chiou, Swarup Bhunia, and Kaushik Roy. Synthesis of application-specific highly-efficient multi-mode systems for low-power applications. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10096, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Jason Cong and Wei Jiang. Pattern-based behavior synthesis for FPGA resource reduction. In *FPGA '08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 107–116, New York, NY, USA, 2008. ACM.

[6] Bertrand Le Gal and Emmanuel Casseau. Automated multimode system design for high performance DSP applications. In *EURASIP*, 2009.

[7] Bita Gorjiara and Daniel Gajski. Automatic architecture refinement techniques for customizing processing elements. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 379–384, New York, NY, USA, 2008. ACM.

[8] Mike Hutton, Jay Schleicher, David Lewis, Richard Yuan Bruce Pedersen, Sinan Kaptanoglu, Gregg Baeckler, Boris Ratchev, Ketan Padalia, Mark Bourgeault, Andy Lee, Henry Kim, and Rahul Saini. Improving fpga performance and area using an adaptable logic module. In *14th International Conference on Field-Programamble Logic, Antwerp, Belgium*, September 2004.

[9] Lab-sticc. GAUT high-level synthesis tool - from C to RTL. `http://www-labsticc.univ-ubs.fr/www-gaut/`.

[10] Prabhat Mishra and Niki Dutt. *Processor Description Languages*. Morgan Kaufman, 2008.

[11] Nahri Moreano, Edson Borin, Cid C. de Souza, and Guido Araujo. Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(7):969–980, 2005.

[12] Chuck Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *ISSS '98: Proceedings of the 11th International Symposium on System Synthesis*, pages 31–36, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Jennifer Stephenson and Paul Metzgen. Logic optimization techniques for multiplexors. In *Mentor user2user conference*, 2004.

[14] Christophe Wolinski, Krzysztof Kuchcinski, Erwan Raffin, and François Charot. Architecture-driven synthesis of reconfigurable cells. In *DSD*, pages 531–538, 2009.

Glossary

ASAP/ALAP: as soon as possible / as late as possible. Two related DFG scheduling algorithms. In ASAP, calculations are performed as soon as their operands are available. In ALAP scheduling, computation is delayed as long as possible without violating a predefined limit on execution time.

ASIC: application-specific integrated circuit. An integrated circuit that is designed for a specific application.

ASIP: application-specific instruction set processor. A processor with custom instructions designed to improve the performance of a specific application.

AST: abstract syntax tree. The result of parsing code according to a set of grammar rules. An AST describes a program at a purely syntactic level, giving information about the structure but not the semantic of the statements.

Datapath: A set of functional units and interconnects performing a calculation. Can be modelled as a graph, wherein each node represents a functional unit, and each edge an interconnection (wire).

DFG: data flow graph. A graph representing the data dependencies in an application. Each node represents an operation, and each edge represents a data dependency.

DSP: digital signal processor. A processor whose instruction set is adapted to digital signal processing applications. Generally contains instructions to accelerate the execution of FIR filters or Fast Fourier Transforms.

FIR filter: finite impulse response filter. A discrete filter with an output of the form

$$y_n = \sum_{i=0}^{N} b_i x_{n-i}$$

is an Nth-order or $N+1$-tap FIR filter. FIR filters have some good numerical properties: they are inherently stable and they have no feedback, thus rounding errors are not compounded.

FPGA: field programmable gate array. An FPGA consists of many small configurable logic blocks (LUTs), generally capable of emulating any 3 to 6-input boolean function, and a reconfigurable interconnect network. Synthesis for FPGA involves breaking down the design into logic block-sized chunks, deciding where to place them, and interconnecting the different blocks as necessary.

ISE: instruction set extension. A set of custom instructions that are added to an instruction set on top of the standard instructions.

LUT: look-up table. The building block of an FPGA, a reconfigurable logic block that can emulate any boolean function of 3-6 variables.

MPU: multi-operation unit. A functional unit that can perform several operations, such as addition, substraction, logical functions, etc.

RTL: register transfer level. A style of hardware description where hardware designs are represented as a set of registers and a set of functions that calculate the next value of a register as a combinatorial function of the current values of the registers.

SoC: System-on-Chip. A SoC is an integrated circuit containing several discrete elements that were previously on separate chips, such as a processing core, several specialized coprocessors, a communications unit, memory, etc. Integrating them on a single circuit has performance and power efficiency benefits.

Softcore: a processor implementation that is sold as RTL description instead of as a physical chip, allowing the user to integrate it into a design and synthesize it for both FPGA and ASIC.

Tap: in the context of FIR filters, a tap is a coefficient.

VHDL: a synthesizable hardware description language that represents hardware designs at Register Transfer Level (RTL).

Appendix

Listing 1: An example of a PSTK processor description

```
processor MIPS{
architecture {
    slot S1 {
        stage FETCH {
            reads  {PCR}
        }
        stage DECODE {
            reads {REG}
            writes {PCR}
        }

        stage EXECUTE {
        }
        stage MEMORY {
            reads {DMEM}
            writes {DMEM}
        }
        stage WRITEBACK {
            writes  {REG}
        }
    }

    register PCR {
        type= PC
        width = 32
        access = { S1 }
    }


    registers REG  {
        depth = 32
        width = 32
        ports = 2R/2W
        access = { S1 }
    }

    memory DMEM {
        depth = 65536
        width = 32
        ports = 1R/1W
    }
}
```

```
instruction_set MIPSI {
    format IType {
        Opcod:6 is opcode
        Regc:5
        Regd:5
        Imm16:16
    }
    format RType {
        Opcod:6 is opcode
        Regdest:5
        Rega:5
        Regb:5
        Func:11
    }

    instruction ADD : ADD : RType {
        opcode = 0
        slots  { S1 }
        computation flow {
            REG[Regdest] = REG[Rega] + REG[Regb];

        }
        program flow {
            PCR = PCR + 4
        }
    }

    instruction MUL : MUL : RType {
        opcode = 22
        slots  { S1 }
        computation flow {
            REG[Regdest] = REG[Rega] * REG[Regb];
        }
        program flow {
            PCR = PCR + 4
        }
    }

    instruction LW : LW : IType {
        opcode = 35
        slots  { S1 }
        computation flow {
            REG[Regd] = DMEM[Regc + Imm16];
        }
        program flow {
            PCR = PCR + 4
        }
    }

    instruction SW : SW : IType {
        opcode = 43
        slots  { S1 }
        computation flow {
            DMEM[Regc + Imm16] = REG[Regd] ;
```

```
        }
        program flow {
            PCR = PCR + 4
        }
    }

    instruction BGTZ : BGTZ : IType {
        opcode = 7
        slots  { S1 }
        computation flow {
        }
        program flow {
            PCR = select(REG[Regc]>0,PCR + 4,PCR + 4 + Imm16)
        }
    }
}
}
```